Programação Orientada a Objectos - Metodologia para o desenvolvimento do trabalho prático

No processo de desenvolvimento de uma aplicação, naturalmente surgem questões e alternativas que é necessário ponderar. Este texto apresenta algumas recomendações que possam orientar as opções a escolher no decorrer deste processo, no contexto de linguagens orientadas a objectos.

A metodologia que este texto apresenta vai sendo exemplificada através de uma aplicação simples que já foi desenvolvida nas aulas práticas.

Exemplo: Pretende-se uma aplicação para representar um desenho formado por uma colecção de rectângulos com lados paralelos aos eixos coordenados. O desenho tem um nome. Relativamente a cada rectângulo conhecemos o seu canto superior esquerdo (um ponto) e as suas dimensões: largura (medida do lado paralelo ao eixo dos xx) e altura (medida do lado paralelo ao eixo dos yy). Os pontos de um plano são representados pelas suas coordenadas cartesianas x e y. Deve ser possível acrescentar um rectângulo ao desenho, obter a soma das áreas de todos os rectângulos, obter o conjunto de rectângulos cujo canto superior esquerdo esteja num dado ponto e obter uma representação textual do conteúdo do desenho.

1. Como começar?

Implementar e testar um subconjunto básico de funcionalidades e expandir passo a passo.

Para desenvolver uma aplicação, não é necessário planear à partida todas as funcionalidades em pormenor. Tendo um conhecimento geral dos objectivos da aplicação, é mais flexível começar por analisar, implementar e testar um subconjunto das funcionalidades (incluindo a interacção com o utilizador para ser possível testar).

Num **desenvolvimento iterativo**, em cada iteração seguinte, o objectivo é ampliar o conjunto de funcionalidades, pensar numa solução para o problema parcial assim definido, implementar e testar.

Neste exemplo, na primeira implementação da solução poderemos considerar apenas as seguintes funcionalidades:

- Acrescentar um rectângulo ao desenho
- Obter uma representação textual do conteúdo do desenho

Depois de implementar e testar estas funcionalidades, seriam acrescentadas e testadas, passo a passo, as restantes funcionalidades.

Neste desenvolvimento iterativo podem considerar-se adaptações ou alterações ao subconjunto já implementado da solução, para responder a problemas que é preciso resolver

ou apenas porque pensando num problema um pouco expandido se encontra uma solução mais adequada.

Em muitas situações existe um conjunto de requisitos fornecidos pelo cliente ou professor. No entanto, raramente os requisitos são completos, e é sempre necessário analisar o problema para deduzir os restantes.

2. Como encontrar as classes que representam a lógica do problema?

Analisar os conceitos envolvidos.

Numa abordagem inicial procuram-se as classes que representam os conceitos relacionados com os objectivos da aplicação, os conceitos do mundo real envolvidos (Pessoa) e eventos (acções que se materializam como informação). Por exemplo, se considerarmos um programa para gestão de stock, poderíamos ter conceitos directamente relacionados com a natureza do problema: Encomenda, conceitos do mundo real: Pessoa (cliente), e eventos: Venda e Pagamento. Esta análise permite começar a resolver o problema, identificando partes menores mais facilmente abordáveis, ajuda também a identificar e relacionar a informação relevante ao programa (sendo este um aspecto muito importante).

O texto relativo à aplicação apresentada como exemplo refere conceitos como:

- Desenho
- Nome
- Rectângulo
- Canto superior esquerdo, ponto
- Largura
- Altura

E funcionalidades como:

- Acrescentar um rectângulo ao desenho
- Obter a soma das áreas de todos os rectângulos
- Obter o conjunto de rectângulos cujo canto superior esquerdo esteja num dado ponto
- Obter uma representação textual do conteúdo do desenho

Entre estes conceitos poderemos identificar classes e atributos de classes (Figura 1).

2.1. Como decidir se um conceito deverá ser considerado classe ou apenas um atributo de uma classe?

Se um determinado conceito tiver significado real para além de um número ou texto, deverá considerar-se uma classe. Por exemplo, ponto (canto superior esquerdo) representa um ponto no plano definido através das suas coordenadas. O seu significado não se resume a um número. O conceito de ponto deve ser considerado uma classe: classe Ponto.

Outra indicação de que um conceito deva ser uma classe e não um mero atributo é a existência de comportamentos autónomos que esse conceito exibe. Por exemplo, um rectângulo produz um valor que é a sua área, se lha perguntarem. O conceito de rectângulo deve ser considerado uma classe: classe Rectangulo.

A largura e altura dos rectângulos são apenas números, o nome do desenho é apenas um texto. Estes conceitos deverão ser considerados atributos.

2.2. Como se relacionam as classes?

Normalmente existem associações entre as classes: as classes relacionam-se umas com as outras. Este relacionamento responde à necessidade que os objectos de uma classe têm acerca de conhecer ou usar objectos de outras classes.

O relacionamento entre classes pode ter significados diversos. Por exemplo:

- Os objectos da classe Desenho contêm objectos da classe Rectangulo. A destruição de um objecto da classe Desenho implica a destruição dos objectos da classe Rectangulo que o objecto da classe Desenho contém. Esta é uma associação da qual é preciso manter a memória: é uma associação persistente.
- Os objectos de uma classe TurmaPratica "vêm" (agregam) objectos da classe Aluno, mas os objectos da classe Aluno existem independentemente da existência dos objectos da classe TurmaPratica. A destruição de um objecto da classe TurmaPratica não implica a destruição dos objectos da classe Aluno que o objecto da classe TurmaPratica agrega. Terminando a existência de uma turma prática os alunos continuam a pertencer a outras turmas, a disciplinas, à escola, têm uma existência independente. Esta é uma associação da qual é preciso manter a memória: é uma associação persistente.
- Os objectos da classe A usam temporariamente objectos da classe B como parâmetros de funções ou variáveis locais de funções. A classe Desenho, num dos seus métodos, usa um objecto da classe Ponto, como parâmetro, na pesquisa dos rectângulos que tenham o canto superior esquerdo coincidente com esse ponto. Não é preciso manter a memória desta associação entre Desenho e Rectangulo: não é uma associação persistente.

Devem ser consideradas as associações persistentes. No exemplo que está a ser analisado, devem considerar-se associações:

- Desenho – Rectangulo: o desenho tem uma colecção de rectângulos
- Rectangulo – Ponto: o rectângulo tem um ponto que é o seu canto superior esquerdo.

Figura 1 Classes, atributos e associações

3. Responsabilidades das classes

É preciso reflectir sobre o que cada classe representa no problema, quais as suas responsabilidades e colaborações (quais os outras classes com que colabora).

A responsabilidade reflecte-se na funcionalidade que deverá existir na classe. As colaborações remetem para a forma como os objectos dessa classe dependem e usam objectos de outras classes.

As responsabilidades poderão ser de dois tipos: fazer e conhecer.

Responsabilidades do tipo **fazer** incluem:

- fazer alguma coisa como criar um objecto ou calcular um valor
- iniciar uma acção noutros objectos
- controlar e coordenar acções noutros objectos

Responsabilidades do tipo **conhecer** incluem:

- conhecer os seus dados privados
- conhecer os objectos com que se relaciona
- conhecer que informação deve obter e calcular

4. Qual a classe que deve criar um determinado objecto?

De um modo geral, a classe que tem com esse objecto uma relação de composição.

Um dos problemas que se coloca será decidir onde deverá ser criado um determinado objecto. É uma responsabilidade do tipo **fazer**. Deve ser atribuída a B a responsabilidade de criar um objecto da classe A se uma (ou mais) das seguintes condições se verificar:

- B "contém" ou tem uma relação de composição com A.
- B grava A.
- B utiliza A de forma muito próxima.
- B tem os dados para construir A.

Por exemplo, no exemplo do desenho, qual é a classe que deve criar um objecto do tipo Rectangulo?

Um objecto do tipo Desenho contém objectos do tipo Rectangulo. Os rectângulos só existem no contexto do desenho. Quando um desenho deixa de existir, os seus rectângulos também deixam de existir. A relação entre Desenho e Rectangulo é de composição. A classe Desenho vai ser responsável pela criação e destruição dos objectos do tipo Rectangulo.

5. Com que critério devem ser atribuídas responsabilidades às classes?

Uma responsabilidade deve ser atribuída à classe que tem a informação necessária para a realizar - *Encapsulamento*

Na aplicação que representa um desenho formado por um conjunto de rectângulos, a que classe se deverá atribuir a responsabilidade de calcular a soma das áreas dos rectângulos? Esta responsabilidade deve ser atribuída à classe que contém todos os rectângulos, ou seja, à classe Desenho. A classe Desenho tem a informação necessária para cumprir esta responsabilidade.

```
class Desenho{
       vector<Rectangulo> rectangulos; // coleccao de rectangulos
       // . . .
       // retorna a soma das areas
       int somaDasAreas()const;
       // . . .
};

int Desenho::somaDasAreas()const {
    int soma = 0;
    for (auto & r:rectangulos) {
        soma += " área do rectângulo r "
    }
    return soma;
}
```

Para cumprir esta responsabilidade, é necessário calcular o valor da área (largura x altura) de cada um dos rectângulos. A classe que tem a informação necessária para cumprir esta responsabilidade é a classe Rectangulo.

```
class Rectangulo {
      // . . .
      int largura;
      int altura;
      // . . .
      // calcula e retorna a area
      int calculaArea()const;
      // . . .
};

int Rectangulo::calculaArea()const {
    return largura * altura;
}
```

A classe Desenho deve delegar o cálculo da área de cada rectângulo na classe Rectângulo:

```
int Desenho::somaDasAreas()const {
    int soma = 0;
    for (auto & r:rectangulos) {
        soma += r.calculaArea();
    }
    return soma;
}
```

Exemplo de uma <u>má solução</u>:

```
int Desenho::somaDasAreas()const {
    int soma = 0;
    for (auto & r:rectangulos) {
        soma += r.getLargura()*r.getAltura();
    }
    return soma;
}
```

Esta solução esvazia a classe Rectangulo das responsabilidades inerentes aos dados que contém, sobrecarregando desnecessariamente a classe Desenho.

Além de esvaziar a classe Rectângulo, viola os preceitos do encapsulamento, tornando visíveis os pormenores internos de implementação da classe Rectangulo e fazendo com que a classe Desenho fique dependente desses pormenores (se a classe Rectangulo for actualizada, terá também que se actualizar a classe Desenho, o que não deveria acontecer).

6. Deve uma classe abranger em pormenor muitas e variadas funcionalidades?

Uma classe deve ter um número de métodos relativamente pequeno com funcionalidade muito relacionada – deve ter alta *Coesão*.

As classes com funcionalidades muito variadas ou com muitas linhas de código têm os seguintes problemas:

- São difíceis de compreender
- São difíceis de reutilizar
- São de difícil manutenção
- São muito afectadas por alterações do software.

Estas classes ou representam um vasto conjunto de responsabilidades ou assumem funcionalidade que deveriam delegar noutras classes.

A coesão mede o grau com que as operações de um elemento de software estão funcionalmente relacionadas e focadas. Uma classe não deve assumir demasiado trabalho. Deve colaborar com outras classes para partilhar o esforço, para não ter demasiado código. As classes com **alta coesão** são mais fáceis de manter, compreender e reutilizar.

Perante uma classe com demasiadas funcionalidades é conveniente:

- verificar se as responsabilidades das classes estão bem distribuídas (encapsulamento);
- considerar a extracção de uma parte coerente de atributos e responsabilidades para a formação de uma nova classe.

7. **Devem ser incluídas no mesmo método as funcionalidades de fazer uma acção e retornar informação correspondente?**

Não: de um modo geral, deve ser adoptado o princípio de *Command-Query Separation* (um método pode ser um comando ou uma *query*, mas nunca ambos).

Na classe Desenho estão as funções acrescentar () para acrescentar um rectângulo ao desenho e getQuantos() para o obter o número de rectângulos.

// estilo #1

```
class Desenho{
       // . . .
       // acrescenta um rectangulo
       void acrescentar(int x, int y, int larg, int alt);
       // . . .
       // retorna o número de rectangulos
       unsigned int getQuantos()const;
       // . . .
};

void Desenho::acrescentar(int x, int y, int larg, int alt) {
       rectangulos.push_back(Rectangulo(x, y, larg, alt));
}

unsigned int Desenho::getQuantos()const {
       return rectangulos.size();
}
```

Por que razão não se deve optar por escrever apenas uma função que acrescente e retorne o número de rectângulos com que fica o desenho?

// estilo #2: <u>porque não está bem</u>?

```
class Desenho{
       // . . .
       // acrescenta um rectangulo e retorna quantos ficam
       unsigned int quantosFicamAcrescentandoUm(int x = 0, int y = 0,
                                                int larg = 1, int alt = 1);
       // . . .
};
unsigned int Desenho::quantosFicamAcrescentandoUm(int x, int y,
                                                  int larg, int alt){
       rectangulos.push_back(Rectangulo(x, y, larg, alt));
       return rectangulos.size();
}
```

Se usarmos o método quantosFicamAcrescentandoUm() do estilo#2 para consultar o número de rectângulos do desenho, é acrescentado um novo rectângulo. Existe um efeito lateral da consulta que altera a informação. Assim, obtemos a informação acerca no novo estado do objecto, mas não ficamos a saber como estava o objecto quando invocámos o método. É melhor ter uma forma que permita consultar o objecto sem o modificar. Por esta razão, o estilo #2 não corresponde a uma implementação correcta.

Segundo o princípio de *Command-Query Separation*, cada método deve ser apenas de um dos dois tipos seguintes:

- ***Command*** - executa uma acção (actualização, coordenação, por exemplo), podendo alterar a informação dos objectos. Não deve ser um método para consulta de informação. O tipo de retorno deve ser void, ou quando muito, pode retornar um valor lógico indicando o sucesso ou fracasso da operação.
- ***Query*** - executa uma consulta retornando informação, sem fazer qualquer alteração. Normalmente é um método const.

8. Como organizar a lógica da aplicação em articulação com a interacção com o utilizador?

Criando um módulo que representa a lógica independente do módulo de interacção com o utilizador.

As classes que integram uma aplicação deverão estar organizadas de acordo com os seguintes módulos:

- **Interface** - módulo que implementa a interacção com o utilizador (menus, comandos envolvendo leitura do teclado e escrita para o ecrã).
- **Lógica da aplicação** - módulo que implementa a gestão da informação, os algoritmos, recebendo os dados necessários do interface e retornando-lhe os resultados.

As classes que fazem parte da lógica da aplicação não devem depender do interface.

As classes do interface não devem conter lógica da aplicação, recebem ordens do utilizador, delegam a sua execução no módulo da lógica e consultam este módulo para mostrar a informação necessária.

9. Qual é o primeiro objecto, para além da camada de interacção com o utilizador, que recebe e coordena uma funcionalidade de natureza lógica?

É um objecto que representa e encapsula toda a lógica.

As classes responsáveis pela interacção com o utilizador não devem conter lógica. O utilizador indica, por exemplo, que pretende acrescentar um rectângulo ao desenho ou obter a soma das áreas dos rectângulos. As classes responsáveis pela interacção delegam estes pedidos às classes responsáveis pela lógica, para que sejam executadas as funcionalidades correspondentes. Deve existir um primeiro objecto das classes responsáveis pela lógica que encapsule toda a lógica e que tenha a responsabilidade de receber e encaminhar o pedido vindo da camada de interacção com o utilizador.

No exemplo em análise, o objecto da classe Desenho teria esta responsabilidade. Normalmente o objecto que encapsula toda a lógica deve delegar trabalho a outros objectos. Coordena e controla a actividade, não executando, ele próprio muito trabalho.

10. Como implementar uma funcionalidade que varia conforme o tipo do objecto que a invoca?

As funcionalidades com significado geral idêntico que variam conforme o tipo de um objecto, devem ser implementadas por funções polimórficas - *Polimorfismo*.

Exemplo: O desenho era formado por um conjunto de rectângulos. Tornou-se necessário considerar outras figuras geométricas, como por exemplo círculos. O desenho deverá poder integrar rectângulos e círculos. Uma das funcionalidades do desenho é obter a soma das áreas das figuras geométricas que o integram. A fórmula para o cálculo da área varia conforme o tipo de figura geométrica.

Exemplo de uma má solução

```
if (tipo == "rectangulo") {
       // cálculo da área com a fórmula da área do rectângulo
       // . . .
} else if (tipo == "circulo") {
       // cálculo da área com a fórmula da área do círculo
       // . . .
}// etc.
```

Perante a situação de lidar com funcionalidades que variam conforme o tipo de um objecto, coloca-se a questão da flexibilidade com que um programa pode ser alterado para se adaptar a novos requisitos. Se se utilizar if-then-else ou switch-case, quando surge uma nova variação a considerar (um novo tipo de figura), é necessário intervir ao nível das instruções if ou switch (interior das funções), em todos os lugares em que a nova variação esteja em causa.

A decisão da acção a realizar não deve resultar da avaliação feita através de uma lógica condicional, usando if ou switch. Os conceitos de rectângulo e círculo são variações de uma figura, pelo que o cálculo da área deve ser uma operação (função) polimórfica.

Deve criar-se uma classe derivada de uma classe base quando:

1. A classe derivada tem atributos adicionais
2. A classe derivada tem associações adicionais
3. A classe derivada tem funcionalidades adicionais ou comportamentos diferentes relativos a funcionalidades que a classe base ou outras derivadas têm.

Os critérios 1 e 3 aplicam-se ao caso dos diferentes tipos de figuras. Cada tipo de figura tem os atributos necessários para o cálculo da sua área (largura e altura para o rectângulo, raio para o círculo). Também a maneira de calcular a área tem uma fórmula específica para cada um destes tipos de figuras geométricas.

Nesta situação, é possível organizar uma hierarquia, em que Figura é a classe base que representa um conceito genérico, e RectanguloFigura e CirculoFigura são classes derivadas que representam conceitos mais especializados.

O nome de cada classe derivada deve terminar com o nome da classe base. Por exemplo RectanguloFigura em vez de Rectangulo. O nome RectanguloFigura sugere mais claramente que representa uma espécie de Figura.

Em cada tipo (classe) que tem o comportamento variável, deve ser criada uma função que implemente a sua versão da acção a realizar. Na classe RectanguloFigura e CirculoFigura varia a maneira de calcular a área. A operação (função) polimórfica pode então chamar-se, por exemplo, calculaArea(). Deve criar-se uma função abstracta calculaArea(), na classe base, Figura, e uma função calculaArea() em cada classe derivada, cada uma delas com a lógica correspondente à classe a que pertence.

Se for preciso expandir o programa de maneira a considerar um novo tipo de figura, basta criar uma nova classe derivada de Figura que vai ter os atributos necessários ao cálculo da área e a função calculaArea() que implementa a sua própria fórmula de cálculo da área.

```
class Figura {
	Ponto ponto; // ponto de referencia
public:
	Figura(int x, int y);
	// . . .
	// calcula e retorna a area
	virtual double calculaArea()const = 0;
};


class RectanguloFigura: public Figura {
	int largura;
	int altura;
public:
	RectanguloFigura(int x , int y , int larg , int alt );
	// . . .
	// calcula e retorna a area
	virtual double calculaArea()const;
};


class CirculoFigura : public Figura {

	static const double PI;
	int raio;
public:
	CirculoFigura(int x, int y, int r);
	// . . .
	// funcoes set

	// calcula e retorna a area
	virtual double calculaArea()const;
};
```

A classe Desenho tem agora uma colecção de figuras que podem ser rectângulos ou círculos (Figura 2). É preciso repensar a estrutura de dados que representa esta colecção.

```
class Desenho{
	vector<Figura *> figuras; // coleccao de figuras
	string nome;// nome do desenho
public:
	// . . .
};
```

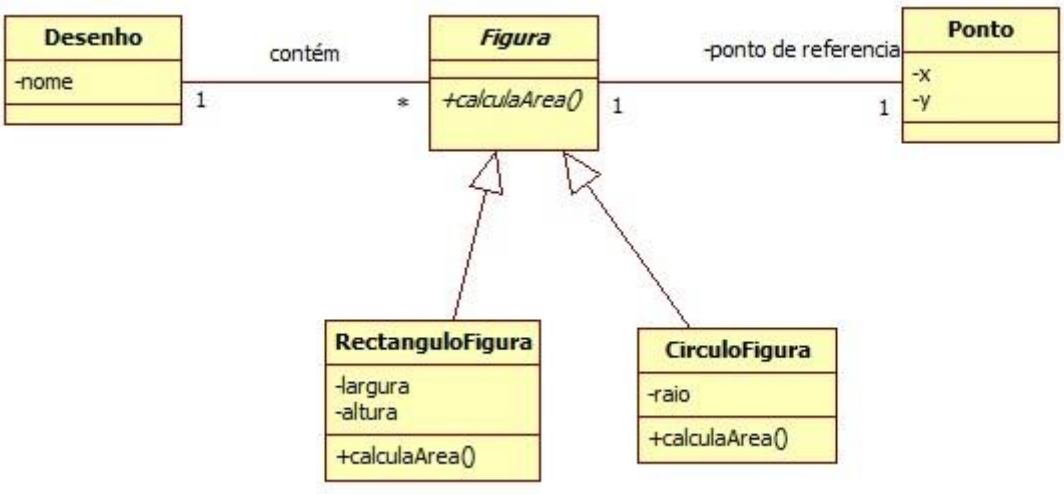

Figura 2 Colecção polimórfica de figuras

Bibliografia

Larman, C., *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*.3rd. ed. 2005: Pearson.